Llamas on the Web: Memory-Efficient, Performance-Portable, and Multi-Precision LLM Inference with WebGPU

Reese Levine
UC Santa Cruz

Rithik Sharma
UC Santa Cruz

Nikhil Jain
UC Santa Cruz

Abhijit Ramesh
UC Santa Cruz

Zheyuan Chen
UC Santa Cruz

Neha Abbas
UC Santa Cruz

James Contini
UC Santa Cruz

Tyler Sorensen
Microsoft Research
UC Santa Cruz

Abstract

Running language models in the browser presents a unique opportunity to build efficient, private, and portable AI applications, but requires contending with constrained memory availability and heterogeneous hardware targets. To realize this opportunity, we present Llamas on the Web (LlamaWeb), a WebGPU backend for llama.cpp that enables memory-efficient and performance-portable LLM inference across a wide range of model weight formats in the browser. Our design significantly reduces memory overhead through static memory planning and efficient model loading, addresses cross-device variability through a tunable kernel library, and introduces templated GPU kernels that support performant implementations of numerous quantization formats, enabling broad model support and extensibility to new formats.

We evaluate LlamaWeb on 16 devices from 8 vendors, collecting data from 10 language models and four model weight formats. We compare LlamaWeb against existing browser-based LLM frameworks and find that LlamaWeb requires 29–33% less memory across several combinations of device, browser, and operating system. We also evaluate LlamaWeb's performance against these frameworks and find that it increases decode throughput by 45–69% across four GPUs from separate vendors. In addition, we compare LlamaWeb's performance against other llama.cpp backends, where it is competitive with and even beats vendor-specific backend performance on some devices.

1 Introduction

The capabilities of (smaller) large language models (LLMs) are improving rapidly [5, 58], with many model developers releasing open-weight models that are explicitly designed for edge deployments [1, 55, 60]. Running models locally can improve inference latency and energy efficiency [27], especially as AI datacenter power usage continues to grow [30, 59]. Additionally, local models provide privacy benefits, with a recent study finding that many frontier AI companies retain and utilize user conversations for training by default and lack transparent privacy policies [33].

Capitalizing on these benefits, inference engines designed to run on consumer and edge devices like llama.cpp [14] and MLX [23] have risen in popularity over the past few years. These engines often ship with vendor-specific GPU backends, e.g., MLX is optimized for Apple GPUs, require expert knowledge to setup correctly, and must be installed separately on a user's system, e.g., on the command line or through an app. On the other hand, web browsers present an attractive choice for local LLM deployments due to their integration into many users' daily tasks [69] as well as their ease of use and lack of external software dependencies. New frameworks like WebGPU [63] enable GPU acceleration within the browser, opening up opportunities for interactive, low-latency usage of language models. Several existing frameworks such as Transformers.js [25], WebLLM [57], and wllama [48] target browser-based inference, with Transformers.js and WebLLM supporting GPU acceleration through ONNX Runtime [44] and MLC-LLM [46], respectively.

Table 1: Comparison of LlamaWeb (this work) to other browser-based LLM inference frameworks on key metrics.

	LlamaWeb	WebLLM	Transformers.js
Available Models[1]	**177,691**	400	41,632
Memory Usage[2]	**Baseline**	+49%	+41%
Performance[3]	**Baseline**	-35%	-41%
Weight Formats[4]	**23**	6	7

[1] Number of compatible models available in May 2026, measured using publicly accessible Hugging Face repositories (App. A).
[2] Geometric mean comparison of normalized peak browser memory usage across several configurations during inference of an `f16` Llama3.2 1B model (Sec. 5).
[3] Geometric mean comparison of normalized decode tokens per second of the same Llama3.2 model across 4 GPUs (Sec. 6).
[4] Number of supported floating-point and quantization formats available for inference in WebGPU (App. B).

However, existing browser-based systems are limited by several issues:

- *Memory Inefficiencies*: Existing browser inference engines dynamically allocate GPU memory and load weights into WebGPU inefficiently, leading to slowdowns as memory grows over the course of execution and even crashes as some browsers enforce hard-caps on memory per-tab.
- *Lack of Performance-Portable Design:* Existing frameworks have limited kernel specialization for different devices, a critical limitation as WebGPU provides *functional* portability but not performance portability. Maximizing performance on the diverse hardware targeted by browsers motivates the need for interfaces that allow kernels to be tuned independently and easily integrated into inference engines.
- *Limited Quantization Support:* To fully realize the opportunity of running models locally, inference engines must support diverse quantization formats that allow upstream model developers to experiment with strategies like quantization aware training and dynamic weight quantization. Kernels that operate on quantized models must be performant while allowing new formats to be added as needed.

Taken together, these limitations hinder the adoption of cross-platform and browser-based LLM inference, motivating the need for new research into techniques to overcome them.

1.1 LlamaWeb

In this work, we introduce LlamaWeb, a WebGPU backend for llama.cpp that runs many models supported by llama.cpp with GPU acceleration in the browser. Our approach is guided by three key constraints: (1) memory usage must be minimal and statically allocated, reducing overhead and allowing useful models to run in the browser on edge devices while avoiding unexpected slowdowns and crashes, (2) kernel libraries must be flexible and extensible to support performance portability across WebGPU's heterogeneous hardware targets, and (3) support for diverse quantization formats needs to be a first-class concern.

By building within llama.cpp, LlamaWeb also benefits from an active open-source ecosystem; for example, the first row in Tab. 1 shows the number of models available in llama.cpp's GGUF format on Hugging Face, as compared to models available in other WebGPU frameworks' formats. In turn, our work extends this ecosystem to the browser, enabling the broad collection of models, quantization strategies, and runtime features developed for llama.cpp to execute efficiently on WebGPU-enabled devices without requiring platform-specific native deployments.

Memory-Efficient Local Inference. A central design goal of our system is minimizing memory overhead during inference. To this end, we statically allocate all memory necessary to run the model at startup, including intermediate data structures for operations like FlashAttention and a fixed-size slotted memory region used for kernel parameters. We also optimize model loading and data movement, avoiding unnecessary materialization of model weights and streaming them asynchronously to GPU memory. In Sec. 5, we compare memory consumption across multiple devices and two browsers (Chrome and Safari) and show that these strategies reduce peak memory usage by 29–33% on average compared to existing browser-based inference frameworks.

Performance-Portable Kernel Library. Maximizing performance across diverse GPU architectures requires adaptable kernel implementations. We develop a kernel library enabling templated shader generation with device-specific optimization and performance-portable parameters. For example, parameters like tile sizes for matrix multiplication can be specialized for different devices or use performance-portable values. Kernels can also incorporate advanced features like subgroup operations or subgroup matrix instructions when supported, while falling back to more portable implementations otherwise. At the execution level, we implement efficient grouping of kernel submission to the GPU and avoid unnecessary CPU-GPU synchronization.

In Sec. 6, we evaluate this system across a diverse set of models and devices, running 10 models across 16 devices from 8 vendors. To our knowledge, this is the largest cross-device, cross-model evaluation dataset collected within a single inference engine framework, and is the only WebGPU dataset that includes mobile devices. We compare our performance against native backends, showing how our kernels achieve portable performance and even beating the performance of native backends in some instances. We further compare our performance against WebLLM and Transformers.js, with LlamaWeb outperforming these frameworks by 54–69% on average during decode. However, during prefill, LlamaWeb underperforms these frameworks by 21–51% during prefill in the browser, which highlights room for future improvement.

Quantization-Aware Shader Design. Supporting the wide variety of quantization formats used in llama.cpp requires careful co-design of kernels and quantized data layouts. Therefore, we develop templated kernels that integrate various dequantization routines directly into computation, e.g., by dequantizing into shared memory or registers while performing row-column reductions. Our approach allows us to implement many of llama.cpp's model weight formats and far more than other frameworks, as shown in the last row of Tab. 1. Our routines are also reusable; for example, the same logic is used when dequantizing weights for matrix multiplication and when accessing KV-cache entries during FlashAttention. To evaluate the effectiveness of our design, we evaluate a Llama3.2 model across multiple quantization levels on a range of devices (Sec. 7), analyzing performance and efficiency trends.

Contributions. In summary, our contributions are:

- A memory-efficient inference engine implementation in WebGPU that maximizes the number of models that can be run on memory-constrained devices and maintaining performance and stability across various platforms (Sec. 3.1).
- A flexible and extensible shader library for LLM inference that allows for external tuning of important parameters, specializes shaders based on available features, and is powered by an optimized scheduling runtime (Sec. 3.2).
- A set of core LLM kernels, namely matrix operations, that are carefully co-designed with llama.cpp's numerous quantization formats, enabling the deployment of models optimized for different sizes and allowing easy integration of emerging quantization methods (Sec. 3.3).

Our core WebGPU backend, which currently consists of 8,470 lines of C++ and 44 kernel template files with 11,338 lines of code, supports 72 core ML operators and 23 data formats, including 32 and 16-bit floating point types and 21 quantization formats. All our code is open-source and has been integrated into llama.cpp codebases through over 100 pull requests, and we look forward to seeing how the community builds on our work going forward.

2 Background

2.1 Language Model Inference

Modern large language models (LLMs) are predominantly autoregressive, where tokens, ranging from a single character to an entire word, are sequentially generated conditioned on previously generated outputs. The main class of such models are transformer-based architectures [62], which rely on the self-attention mechanism to process and generate new tokens. Alternative approaches have been proposed to address the quadratic increase in computation and memory as sequence length increases in attention mechanisms, including architectures based on state space models [21] and gated convolutions [2].

Most modern models' inference runtime is dominated by linear algebra operations such as matrix–matrix and matrix–vector multiplications. This runtime can be broken down into two distinct phases: (1) *prefill*, where an entire prompt is processed, and (2) *decode*, where new tokens are generated sequentially. During

prefill, dense matrix multiplication is used to process all tokens in the prompt, while during decode matrix-vector multiplication is often used to process a single token. A model's representation of previously processed tokens in the attention mechanism can be stored in a *KV-cache*, which is utilized to compute attention scores when generating new tokens. The complexity of attention has led to the development of hardware-efficient mechanisms like FlashAttention [8], which fuses multiple matrix multiplications with a softmax normalization to avoid multiple round-trips to DRAM on GPUs.

Inference engines such as llama.cpp [14] provide a concrete implementation of these ideas. In llama.cpp, model weights are stored in a custom binary format (GGUF) which encodes tensors containing model weights, along with metadata, and can be distributed through model repositories like Hugging Face. The model architectures themselves are described within the llama.cpp codebase. At execution time, a dynamic graph of tensor operations is constructed, ranging from metadata-only operations that do not access or move the underlying data, e.g., reshape and transpose, to materializing operations that do move or compute on tensor data, e.g., matrix multiplication. This directed acyclic graph (DAG) is then linearized into a topological ordering and executed by different backends.

The llama.cpp codebase contains various backends that allow models to be executed on diverse hardware, including CPUs, GPUs, and NPUs. Most existing backends target devices from specific vendors, including a CUDA backend for NVIDIA GPUs, a Metal backend for Apple GPUs, and a HIP backend for AMD GPUs. Cross-platform backends like Vulkan and OpenCL also exist, but no existing backends were designed with the constraints of browser-based environments in mind.

Some inference engines, such as vLLM [34], are specifically designed for multi-GPU and server-based inference. These engines incorporate sophisticated sequence-batching strategies that maximize throughput and avoid GPU under-utilization during the decode phase. While llama.cpp is evolving to support these use-cases as well, our WebGPU implementation is specifically focused on low-latency single-GPU inference, e.g., in the browser, so we do not explore batching in this work.

2.2 Model Quantization

The size of modern language models presents a major barrier to local deployment. Storing model weights in full 32-bit or even 16-bit floating point format leads to large memory requirements, often exceeding the capacity of consumer devices. As a result, quantization techniques have become essential for enabling efficient inference. Quantization reduces the precision of model weights (and sometimes activations) to lower-bit representations, such as 8-bit or 4-bit integers, reducing both memory footprint and bandwidth requirements. Llama.cpp supports a variety of quantization formats, with most following the form:

$$q_i = \text{round}\Big(\frac{x_i - \mu}{s}\Big), \qquad \hat{x}_i = s \cdot q_i + \mu \tag{1}$$

where x_i is the original weight, s is a scaling factor, μ is an optional offset depending on the format, q_i is the computed quantized value, and $\hat{x}_i$ is the resulting dequantized weight. Weights are quantized by grouping them into blocks and choosing scaling factors based on the range of weight values within that block, e.g., by calculating the absolute maximum. The three major quantization formats supported by llama.cpp are:

- **Basic (legacy):** These formats use blocks of 32 weights. Variants like `q4_0` and `q8_0` are *symmetric*, i.e., they do not include offsets, while variants like `q4_1` compute and store offsets per-block.
- **K-quants:** These formats use blocks of 256 weights, which are subdivided into blocks of 32, each symmetric and with their own scaling factor. Scaling factors are themselves quantized into a single super-block scaling factor, further reducing the memory overhead. While compression of quantization constants has a long history in signal processing [15], the design decisions behind llama.cpp K-quants were never explicitly laid out [31]. However, they are similar to the double quantization technique introduced in QLoRA [11].
- **I-quants:** These formats also use blocks of 256 weights, but are inspired by vector quantization, including methods like QuIP# [61], where small groups of weights (usually 8) are encoded as an index to a codebook of reference vectors.

All of these quantization formats are *generic*; that is, they are not designed for particular hardware, but rather dequantization is implemented in software and operations are performed on the resulting values. Along with these generic formats, llama.cpp also supports hardware-native formats like `bf16`, `nvp4`, and `mxfp4`. These formats can also be emulated in software, and the WebGPU backend does support `mxfp4` emulation in order to run OpenAI's gpt-oss-20b model [53], but emulation for other native formats is not currently supported. Recently, a quantization format called `q1_0` was introduced to support the Bonsai 1-bit model family [55], which uses symmetric quantization with a single scale for a block of 128 weights. Due to our quantization-aware kernel design, we were able to easily integrate support for this quantization type with minimal effort, as described in Sec. 3.3.

Despite their widespread use, there is limited formal documentation or literature describing the design and especially the data layout of these quantization formats. Many formats rely on non-contiguous or highly specialized packing schemes that may have been optimized for specific hardware access patterns. Understanding the trade-offs in quantization block layout with respect to vectorization, memory alignment, and GPU execution remains an open area for future work.

The quantization formats in llama.cpp are *weight-only*. However, llama.cpp has also incorporated techniques from other post-training quantization methods like GPTQ [13] and AWQ [36] into its quantization tools, and the llama.cpp community is also working on novel techniques for mixed-precision quantization using llama.cpp's standard formats. Model developers can also perform quantization-aware training [29] using a given quantization format and deploy the resulting model with llama.cpp.

2.3 WebGPU Programming Model

WebGPU is a modern cross-platform graphics and compute API designed to provide low-level access to GPU hardware within web browsers. Despite its name, WebGPU can also be used directly by native applications, as some WebGPU implementations are independent libraries that can run outside browser contexts. Like most other

GPU programming models, WebGPU separates host-side control logic from device-side shader execution. While WebGPU supports both graphics and compute use-cases, for this work we focus only on its compute capabilities.

On the device side, kernels (called *shaders* in WebGPU) are written in the WebGPU Shading Language (WGSL), which supports general-purpose compute in a Single Instruction, Multiple Threads (SIMT) execution model. Threads are organized into workgroups, where each thread has access to private registers, workgroups share fast on-chip memory (analogous to shared memory in CUDA), and all threads can access global device memory. Recent extensions to the WebGPU programming model include subgroups (analogous to warps in CUDA), which represent smaller collections of threads that can execute cooperative operations such as reductions or shuffles using registers. Emerging features such as subgroup matrix operations further expose specialized hardware units, e.g., tensor cores, for efficient matrix multiplication.

The WGSL language supports various data-types, including 32 and 16-bit floating points (`f32` and `f16`), signed and unsigned 32-bit integer types (`i32` and `u32`), and vector types. It does not yet support specialized types like `bf16` [18], smaller integer types like `u16` and `u8`, or specialized 4-bit types like `nvfp4` [50] and `mxfp4` [52].

On the host side, applications interact with the GPU (*device*) through a structured API. Kernels are compiled at runtime into *pipelines*, which define both the executable code and the layout of bound resources. Data is stored in *buffers* allocated by the host and organized into *bind groups* which are bound to pipelines according to specified *bind group layouts*. WebGPU follows a deferred execution model; compute workloads are recorded via *command encoders* and *compute passes*, but this work does not execute on the GPU until the command encoder is *finished*, i.e., flushed, into a *command buffer*, which is in turn submitted to the device *queue*. Along with command buffers, work like copying data from one buffer to another can be submitted to the queue. The granularity of queue submission, such as how many operations are grouped into a single compute pass or how many commands are in flight in the queue, can affect performance and stability, and application performance can vary across different WebGPU implementations.

WebGPU is designed as a portability layer over multiple native APIs, with current implementations supporting Metal on macOS/iOS [3], Vulkan on Linux and Android [20], and DirectX on Windows [42]. Many major browsers have their own WebGPU implementations: Dawn [16] is used in Chromium-based browsers, wgpu [40] in Firefox, and WebKit [65] provides its own implementation for Safari. While WebGPU is primarily designed for browsers, implementations like Dawn, which is written in C++, can also be used with native code. Tools like Emscripten [68] enable cross-compilation of C++ to WebAssembly (WASM), a portable low-level bytecode that, like WebGPU, is built to abstract over different hardware and platforms [22]. Therefore, although we mainly focus on inference in the browser in this work, LlamaWeb can also be used in any environment that supports C++ or WASM.

3 Design of LlamaWeb

Figure 1 gives an overview of the llama.cpp inference engine. GGUF files are loaded by llama.cpp and the tensor operation graph is dispatched to backends through a standardized interface. As mentioned, llama.cpp supports numerous backends; our work, the highlighted WebGPU backend, targets GPUs from many vendors and adds support for GPU-accelerated inference in the browser.

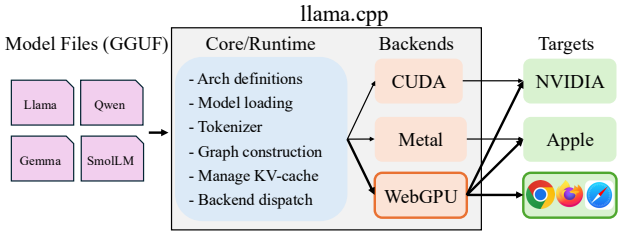


Figure 1: Overview of llama.cpp's core and backend design.

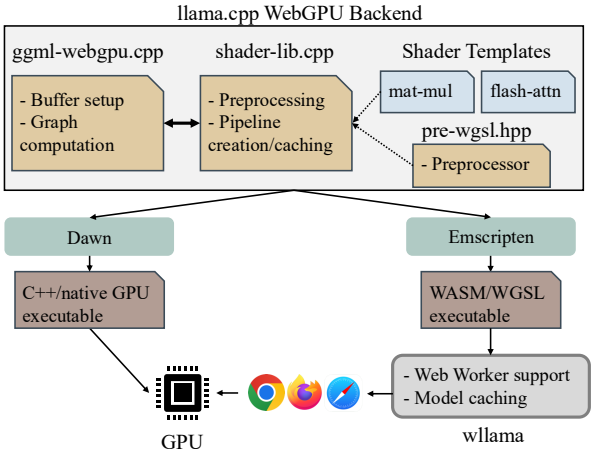


Figure 2: Breakdown of the LlamaWeb llama.cpp WebGPU backend and its different paths for executing on GPUs. The backend can be built for native execution using Dawn as the WebGPU implementation, or built for cross-browser execution with Emscripten.

Figure 2 shows a more detailed diagram of how the WebGPU backend is implemented. The main implementation is split into two files: `ggml-webgpu.cpp`, which implements llama.cpp's backend interface, and `shader-lib.cpp`[1], which handles preprocessing and compilation of kernels into WebGPU pipelines.

As WebGPU does not have a built-in preprocessor, many WebGPU applications have developed ad-hoc preprocessing, e.g., using JavaScript string substitution and concatenation. Motivated by our need for a more standard solution that works within llama.cpp's C++ codebase and is familiar to developers, we built a minimal WGSL preprocessor which we call pre-wgsl [35]. Pre-wgsl is a header-only library that is directly included in the WebGPU backend and can also be used independently either natively or in web environments.

[1]In this paper we use the term kernels to refer to shaders, but in our code we follow the WebGPU convention of calling them shaders.

To run llama.cpp with WebGPU acceleration in the browser, we add WebGPU support to the pre-existing wllama library, which previously only supported running llama.cpp through WASM on the CPU. Wllama includes features like caching models in the browser's Origin Private File System (OPFS) [47] and running WebGPU in a Web Worker to avoid freezing the UI thread. In the rest of this section, we describe in more detail the key design points of LlamaWeb.

3.1 Memory-Efficient Browser Inference

In LlamaWeb, memory is statically allocated at startup, ensuring predictable memory usage independent of prompt or generated output size. To accomplish this, we do the following: (1) implement a static parameter buffer "arena" which avoids dynamic GPU buffer creation and complicated caching logic, (2) calculate up-front any extra memory needed for intermediate states, e.g., to implement FlashDecoding [9], and (3) optimize model loading in the browser to avoid redundant memory and large CPU memory allocations. Inherited from llama.cpp, LlamaWeb also utilizes a fixed-size KV-cache which can be allocated before running inference.

Each kernel needs a small number of dynamic parameters, e.g., matrix dimensions, passed in. Unlike in other languages, e.g., using push constants in Vulkan, WebGPU does not yet support a way to directly pass small amounts of data to kernels at runtime, so frameworks like WebLLM and ONNX Runtime maintain pools of small buffers that are dynamically allocated and cached. Though the memory overhead of these pools is small, when pushing against the limits of a device's memory, allocating a new buffer may be the difference between crashing the page or not. Therefore, LlamaWeb allocates a single buffer at startup with enough slots to fit the parameters for a configurable number of kernels. Within the buffer, we rotate through the slots, and we guarantee that parameters are not overwritten until the kernel that relies on them finishes. This simple design also avoids any synchronization logic needed to maintain a buffer pool, including handling asynchronous callbacks to release buffers, and avoids memory fragmentation.

During some operations, e.g., FlashDecoding, several workgroups cooperate on computing attention scores across a single query vector, and per-workgroup results are stored in an intermediate buffer which is reduced by a separate kernel. Also for FlashDecoding, as well as for FlashAttention and sampling operations like `top-k`, we implement optimizations that run several kernels, some of which require intermediate memory storage space. In all cases, LlamaWeb allocates this extra memory *before* the model first runs.

Lastly, we optimize wllama to efficiently load model weights. First, we avoid redundant memory copies by downloading and storing weights in disk (utilizing OPFS) and never materialize them in the WebAssembly heap[2]. Next, we utilize llama.cpp's asynchronous loading interface to reduce the memory footprint of the loading process, using only four 1 MB buffers to stream weights directly from OPFS into WebGPU buffers. In particular, we found that Safari has especially strict memory usage limits, so these changes enabled us to serve larger, more powerful models with our system.

[2]This is especially important because the WASM heap is grow-only [64], so once memory is allocated by a tab it can't be released during the tab's lifetime.

Table 2: Percentage of time spent in different kernel categories during prefill (512 token prompt) and decode (128 tokens generated) when running LlaMA 3.2 1B (`q4_k_m` format) on an Apple M3 at KV-cache depths of 0 and 2048.

	KV@0		KV@2048	
Kernel Category	**prefill**	**decode**	**prefill**	**decode**
Matrix multiplication	92.6%	0.0%	83.4%	0.0%
Matrix-vector multiplication	0.5%	89.9%	0.3%	80.7%
Attention	4.4%	5.3%	14.1%	15.0%
Normalization/elementwise	2.2%	3.0%	2.0%	2.7%
Other	0.3%	1.8%	0.3%	1.6%

3.2 Kernel Library and Execution Scheduling

The WebGPU backend is separated into two layers: the runtime execution logic and the kernel library. The runtime is responsible for high-level orchestration; it constructs and passes a lightweight context to the library that describes the tensors involved, e.g., shapes, strides, and data types, along with device-specific information such as limits and supported features. Then, given compiled pipelines returned by the kernel library, the runtime handles logic for execution scheduling and grouping operations into compute passes, managing command buffer submission, and coordinating CPU–GPU synchronization.

Given an operation and context, the kernel library first performs a preprocessing step to construct an appropriate kernel variant. This includes decisions such as selecting code blocks based on tensor data formats (e.g., `f32`, `f16`, or quantized formats), choosing between workgroup-level or subgroup-level reductions depending on device features, determining whether an operation must be performed in-place (which affects how buffers are bound to the kernel as WebGPU does not allow buffer aliasing), and configuring tiling or vectorization strategies. Based on these choices, the library generates kernel code and compiles it into a GPU pipeline. To avoid redundant work, compiled pipelines are cached using a key that encodes the information used to specialize the kernel. Alongside the compiled pipeline, the kernel library can also return metadata describing the decisions it made, which can inform subsequent dispatch shapes and scheduling decisions.

This separation of concerns between the kernel library and runtime execution logic means that in the future, the library can independently evolve to support increasingly sophisticated performance-portable strategies, including device-specific tuning, dynamic specialization, and even auto-tuning mechanisms. Currently, we choose performance-portable parameters for operations like matrix multiplication based on a large empirical study, while other operations use heuristically chosen values for parameters like workgroup size based on pilot experiments. Kernel compilation incurs a one-time cost (~1-5 seconds) during the first forward pass on the model, after which execution proceeds using cached pipelines.

Several categories of kernels are necessary to run a full forward-pass on a model. Table 2 shows the percentage of time spent executing each kernel category for a representative benchmarking run. Matrix and matrix-vector multiplication kernels dominate runtime in prefill and decode respectively, while normalization/elementwise

and other operations, including rotary position encoding and tensor data movement, are a small percentage of runtime. As expected, time spent doing attention grows as the KV-cache depth increases.

We now briefly describe our implementations of these different kernel categories. All kernels are validated using llama.cpp's testing infrastructure, which compares the output of running the operation on the GPU to running a reference implementation on the CPU, and ensuring the numerical mean squared error (NMSE) across the output remains under a threshold, commonly $1e^{-7}$. During development, we observed that differences in floating-point rounding behavior while casting from `f32` to `f16` values caused higher NMSE on some devices, e.g., an NVIDIA Tesla T4, so for operations on `f16` tensor data the NMSE threshold is relaxed to $1e^{-6}$.

General Purpose Kernels. Our library includes kernels performing a wide variety of general-purpose tasks on the GPU. For example, it supports many unary, e.g., `floor`, `ceil`, `abs`, and binary, e.g., `add`, `mul`, `div`, operations. It also includes support for important LLM operations like gated linear units (`glu`), rotary position embedding (`rope`), and normalizations, and for sampling[3], operations like `top-k` and `argmax`. To support state space models, the library also includes support for convolutions and GPU prefix scans. Although general operations like these are normally not the largest bottleneck in LLM inference, our library design allows these kernels to be optimized and specialized as necessary, e.g., by using subgroup reductions or vectorized memory accesses.

Matrix Multiplication. As it is one of the most important operations used in LLM inference, the library implements extensive support for matrix multiplication in `f32`, `f16`, and quantized formats. It includes two versions of matrix-matrix multiplication, `reg_tile` and `sg_mat`, which are primarily used during the prefill phase of LLM inference. Both follow a hierarchical tiling strategy; `reg_tile` first tiles into shared memory, then into registers per-thread[4], while `sg_mat` also tiles into shared memory, but then utilizes WebGPU's newer subgroup matrix feature to run cooperative matrix multiplications on specialized hardware like NVIDIA's tensor cores. The subgroup matrix feature is not yet available in stable browser releases, but can be used natively in some WebGPU implementations.

The library also includes a specialized matrix-vector multiplication kernel for the decode phase. This kernel tiles the vector in registers and does a cooperative reduction across each row. When subgroups are available, the kernel is specialized to perform a subgroup reduction, otherwise falling back to a generic shared memory reduction. Each of these kernels exposes tunable workgroup sizes, shared-memory tile sizes, and per-thread or per-subgroup tile sizes.

To select performance-portable defaults, we ran a large empirical study, exhaustively sweeping thousands of configurations across four GPUs from four different vendors over matrix shapes drawn from representative LLM workloads. From this data we derived performance-portable defaults that maximize average performance across the GPUs while minimizing worst-case slowdowns, delivering a 41% kernel-level speedup on average over hand-picked values chosen during development (which was primarily done on systems with Apple GPUs). Pushing this data-driven approach further to more devices, more workload sizes, and more browser configurations is a natural direction for future work (Sec. 9).

FlashAttention. As attention is one of the central operations in LLM inference, our library implements several variants. Rather than materializing the intermediate `QK^T` scores and softmax probabilities, these kernels stream over the KV cache in tiles and maintain the online-softmax state directly inside the kernel. The FlashDecoding implementation is optimized for decode; it maps one query row to one workgroup, keeps the row maximum, exponential sum, and output accumulator local to that row, and iterates over cached K/V tiles. The `tile` path targets larger query chunks, e.g., during prefill, by processing multiple query rows per workgroup and staging Q/K/V tiles in workgroup memory for better reuse. The implementation also contains a subgroup-matrix variant for environments where this experimental WebGPU feature is supported. The backend also supports quantized KV-cache formats such as `q4_0` and `q8_0` by dequantizing K/V blocks while loading them from global to shared memory in the attention kernel.

3.3 Quantization-Aware Kernel Design

As introduced previously, llama.cpp supports a diverse and evolving set of quantization formats, ranging from legacy schemes, e.g., `q4_0`, `q8_0`, to more advanced formats such as K-quants, I-quants, and most recently `q1_0`. Dequantization is primarily needed in operations involving model weights, e.g., matrix-matrix and matrix-vector multiplications. In these operations, weights are stored in compressed formats and must be dequantized on-the-fly during computation. However, the activations in the KV-cache used in attention can also be stored in quantized form, requiring dequantization during attention score computation.

A central challenge arises from the diversity of quantization layouts. Formats differ in block sizes, scaling factors, and bit-packing strategies. WebGPU has limitations on how types like structs are loaded from memory, as well as the required alignment of data types. To achieve better performance and portability, in the kernels all quantization formats are represented as flat buffers of unsigned 32-bit (`u32`) types, with dequantization routines interpreting these values depending on a format specified at compile time[5].

Dequantization routines are integrated directly into kernels, with the particular routine chosen during preprocessing by the kernel library at compile time. For matrix multiplication, which is primarily compute-bound on most GPUs, we adopt a shared-memory tiling strategy; threads within a workgroup collaboratively load quantized blocks, dequantize them into shared memory, and reuse the decoded values across multiple output elements. Beyond weight operations, these routines can also be reused for KV-cache dequantization during attention.

In contrast, matrix-vector multiplication is a memory-bound operation with limited data reuse. During implementation of this operation, we determined that a shared-memory tiling strategy for

[3]We find that offloading single-token sampling to the GPU in WebGPU currently does not improve performance. Optimization here is an area for future work.

[4]During development we observed that using `f16` accumulation for the register-tiling kernel caused some models, e.g., Qwen2.5, to generate incoherent output on Apple M-series GPUs, so we currently use `f32` accumulation for this kernel. Differences in floating-point precision is a well-known issue, and handling it within WebGPU and its applications is not a solved problem.

[5]For efficient memory loading, it would be more beneficial to represent some quantization formats as buffers of `u16`s, but WebGPU does not yet support this type.

Table 3: Models and weight formats used for cross-device evaluation. `llama` is additionally used in the cross-framework, browser-vs.-native, and cross-quantization studies; the four variants in its rows span the cross-quantization study.

Model	Short Name	Type	Weights	Filesize (GB)
LFM2.5-350M [2]	`lfm`	H	`q4_k_m`	0.23
Bonsai-1.7B [55]	`bonsai`	B	`q1_0`	0.25
Gemma3-270M [60]	`gemma3`	T	`q4_k_m`	0.25
Qwen3-0.6B [67]	`qwen3`	T	`q4_k_m`	0.40
Granite4-1B [28]	`granite`	H	`q4_k_m`	0.90
Qwen3.5-2B [56]	`qwen3.5`	T	`q4_k_m`	1.28
SmolLM3-3B [4]	`smollm`	T	`q4_k_m`	1.92
Ministral3-3B [37]	`ministral`	T	`q4_k_m`	2.15
Gemma4-E2B [19]	`gemma4`	T	`q4_k_m`	3.11
Llama3.2-1B [41]	`llama`	T	`q2_k`, `q4_k_m`	0.58, 0.81
			`q8_0`, `f16`	1.32, 2.48

T: Transformer. H: Hybrid SSM. B: 1-bit transformer.

Table 4: Devices used in our evaluation. Chrome is the browser used except on iOS, where Safari is the only WebGPU option.

Vendor	GPUs	Operating Systems	Example Hardware
NVIDIA	3	Linux, Windows	RTX 5080*†, RTX 5070
AMD	1	Linux	RX 7900 XT†
Intel	2	Linux, Windows	Arc B580†, Iris Xe
Apple (macOS)	3	macOS	M4 (32 GB)*†, M2
Apple (iOS)	2	iOS	iPhone 17 Pro Max
Qualcomm	2	Android, Windows	Snapdragon X Elite
Samsung	1	Android	Galaxy S24 (Xclipse)
ARM	1	Linux	Mali (Valhall)
Imagination Tech.	1	Android	PowerVR D-series
Total GPUs	**16**		

* Used in cross-framework memory efficiency eval.
† Used in cross-framework and native performance evals.

matrix-vector multiplication, as appears in other cross-platform portable libraries like CLBlast [49], was not a good fit for the many dequantization routines we wanted to support. Instead, we implement routines that dequantize directly into registers, improving access patterns to quantized memory and allowing the same underlying kernel to support llama.cpp's diverse quantization families.

Our implementation supports the majority of quantization formats available in llama.cpp, including legacy formats, K-quants, I-quants, and `q1_0`. Notably, this support is achieved without requiring format-specific kernel rewrites and enables rapid extensibility: for example, when `q1_0` was introduced alongside the Bonsai model family, we were able to integrate support by adding the new dequantization routine without modifying the surrounding matrix operation implementations.

While our current implementations demonstrate competitive performance, they also reveal opportunities for further optimization. In particular, tighter integration between quantization layouts and GPU execution strategies, e.g., exploiting subgroup operations or specializing for specific bit-width patterns, may yield additional gains. These observations motivate future work on deeper co-design between quantization schemes and hardware-aware kernel generation.

4 Experimental Setup

We evaluate LlamaWeb across a diverse set of models, model weight formats, hardware platforms, and execution environments. Table 3 summarizes the models and weight formats used in our study, while Tab. 4 summarizes the evaluated devices. Our evaluation spans 10 models across 16 devices from 8 GPU vendors, including desktop and mobile hardware. The evaluated models include traditional transformer architectures, recent hybrid state-space models, and a 1-bit transformer variant, allowing us to study the behavior of WebGPU inference across diverse model architectures.

Many llama.cpp quantization strategies do not uniformly quantize model weights across all layers, instead heuristically choosing a layer's weight format. Therefore, llama.cpp model variants are often named after the quantization format that is predominantly used. In particular, several strategies for `q4_k` quantization exist. To standardize our experiments, we specifically use `q4_k_m` model variants for all llama.cpp `q4_k` evaluation.

To support large-scale evaluation in the browser, we develop a benchmarking harness for the browser that automates model loading, warmup runs, repeated execution, and collection of throughput statistics. Not all models run on all devices; for example, on iOS devices Safari tab memory is limited to <500 MB. The full evaluation sweep takes anywhere from ~10 minutes on a high-end discrete GPU to over an hour on low-power mobile GPUs. Across all 16 devices, cumulative time for the 481 benchmark runs in our dataset was approximately 25 hours.

For performance measurements, we separately evaluate both prefill and decode phases of inference. Prefill measurements use an input prompt length of 512 tokens, while decode measurements generate 128 output tokens. To study the impact of attention and KV-cache growth on performance, for some evaluations we measure each workload at KV-cache depths of 0 and 2048 tokens. Each experiment is repeated 5 times and we report averaged results.

All browser-based experiments are executed using WebGPU-enabled browsers on each platform. In our pilot experiments, we observed substantial implementation-dependent performance differences across browsers. Chrome and its underlying Dawn WebGPU implementation consistently provided the highest and most stable WebGPU performance across many platforms, and is therefore used for all reported experiments unless otherwise noted. Safari is used for iOS devices where alternative browser engines are unavailable. Although Firefox supports WebGPU, we observe substantially lower performance, e.g., on an Apple M3 the `llama` model with `q4_k_m` weights runs at ~1 tok/s on Firefox, but ~52 tok/s on Chrome, so we therefore omit Firefox results from our evaluation.

5 Memory Efficiency

Figure 3 shows memory usage while running inference in the browser on the `llama` model with `f16` weights across different frameworks, browsers, and devices. We collected data using LlamaWeb, the WebLLM example webpage [45], and Transformers.js examples [26][6]. Memory usage data was collected by launching a fresh browser instance and using platform-native tools to collect

[6] At evaluation time these examples had not been updated to Transformers.js 4.0, so we manually updated them to ensure we were comparing against the most recent release.

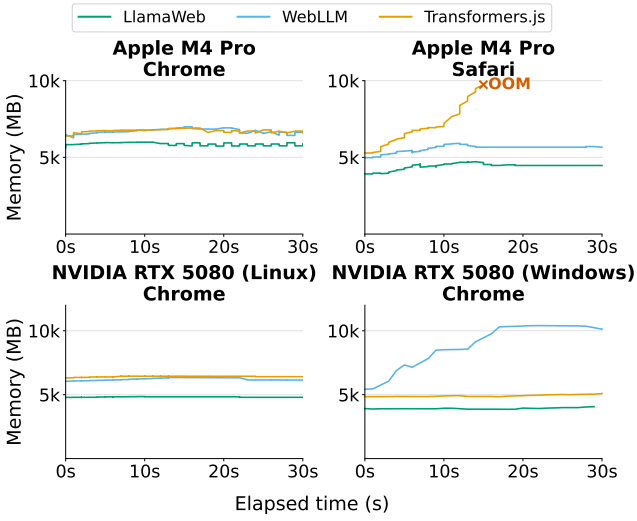
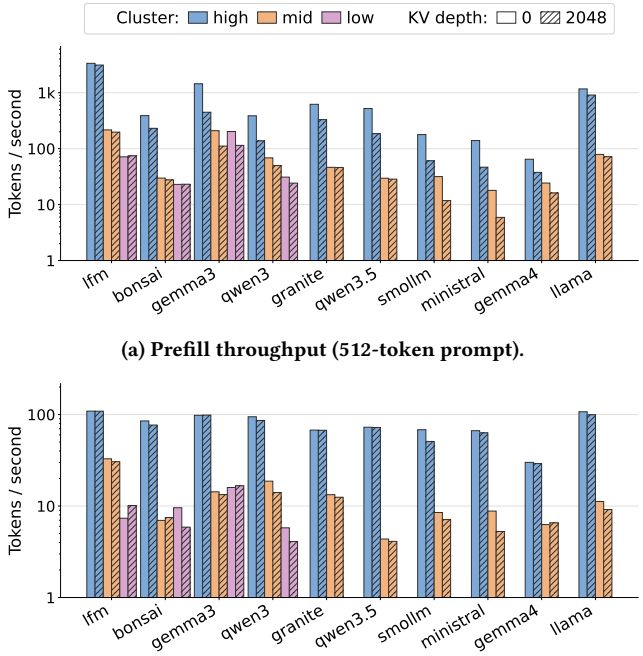


Figure 3: Memory usage for the `llama` model with `f16` weights during inference.



(a) Prefill throughput (512-token prompt).

(b) Decode throughput (128 generated tokens).

Figure 4: Median LlamaWeb throughput across all 10 models (Tab. 3) on 16 GPUs from 8 vendors, grouped into three performance clusters via k-means.

memory usage for the unified tab process for Safari, and the tab process and GPU renderer process for Chrome. Each framework was prompted to generate sufficient decode output for measurement.

In all four cases, LlamaWeb uses the least memory, with a geometric mean of normalized peak memory usage 49% lower than WebLLM and 41% lower than Transformers.js across the four testing configurations. On the Apple M4 Pro GPU, the memory usage of LlamaWeb was 13% better than WebLLM and 16% better than Transformers.js on Chrome. On Safari, this gap grew to 28% better than WebLLM and 59% better than Transformers.js. As the graph shows, the memory usage of Transformers.js on Safari climbed to 10 GB until the tab was eventually killed, meaning that some combination of the Transformers.js codebase and Safari's WebGPU implementation caused a memory leak.

LlamaWeb also used less memory than other frameworks on the NVIDIA RTX 5080. Running Chrome on Linux, LlamaWeb used 24% less memory than Transformers.js and 23% less than WebLLM. Running Chrome on Windows, LlamaWeb used 20% less memory than Transformers.js and 61% less than WebLLM. WebLLM memory usage also climbed to around 10 GB on this configuration, most likely due to a memory leak. While total memory usage varies widely based on operating system, browser, and WebGPU backend, LlamaWeb consistently uses the least memory and does not suffer from memory leaks. Together, these results validate the memory-efficient design of LlamaWeb and show that developers must pay close attention to cross-platform memory usage while building applications using WebGPU.

The inefficiency of Transformers.js's steady-state memory usage can be explained in part by how it creates temporary copies of models into CPU buffers before sending them to GPU buffers, creating an unnecessarily large memory footprint. Similarly, WebLLM temporarily loads the entire model into JavaScript memory before sending it to the GPU, whereas our integration with wllama loads models into GPU buffers directly from OPFS through a small set of transfer buffers.

A last point of comparison is model sizes; llama.cpp's GGUF format and MLC's model files are often similarly sized, while ONNX models are larger due to serialization overhead. WebLLM and ONNX model files are also mandatorily sharded, by design or due to Protobuf's 2 GiB maximum file size [17] respectively. In contrast, GGUF models can be stored as either a single file or sharded across multiple files, giving developers more flexibility in deployment strategies.

6 Performance Evaluations

We evaluate cross-device performance across the 10 models in our study on 16 devices, and evaluate native performance and compare performance of several WebGPU inference frameworks on an NVIDIA RTX 5080, Apple M4 Pro, AMD RX 7900 XT, and Intel Arc B580.

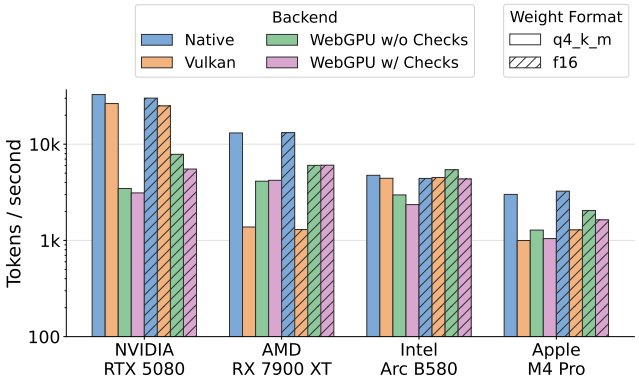
6.1 Performance-Portability Across Models and Devices

To evaluate LlamaWeb's portability, we run all 10 models in Tab. 3 across every WebGPU-capable device in Tab. 4 with `q4_k_m` weights (or `q1_0` for Bonsai) and depths of 0 and 2048 for the KV-cache. To understand how different GPUs behave and their similarities to one another, we group the devices into three clusters via k-means on each device's log-throughput feature vector across all (model,

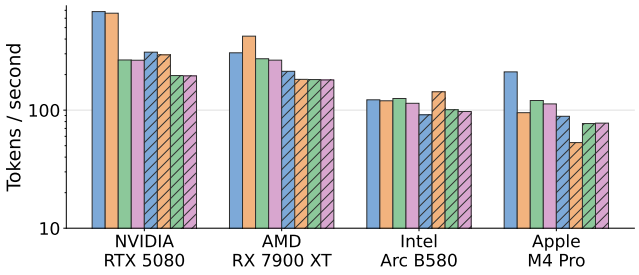
phase, KV-cache depth) measurements. For more details on the clustering strategy, see App. C.

The three clusters end up reasonably representing GPU capabilities and performance: the first cluster (`high`) contains high-end discrete GPUs like the NVIDIA RTX 5080, the second cluster (`mid`) contains integrated and high-end mobile GPUs like an Apple M2 and Galaxy S24, and the third cluster (`low`) contains low-power and efficient GPUs from iPhones and Android devices (Adreno, Mali, PowerVR). To our knowledge, this is the largest cross-device, cross-model evaluation of a single browser-based inference engine, and the first to include mobile devices. LlamaWeb runs every model in our suite on devices in the `high` and `mid` clusters, while GPUs in the `low` cluster were only able to fit the four smallest models (`lfm`, `bonsai`, `gemma3`, and `qwen3`) due to memory constraints.

Figure 4 shows the median prefill and decode tok/s throughput for each model, cluster, and KV-cache depth. Throughput varies by several orders of magnitude due to varied device performance and model size, highlighting how LlamaWeb is able to adapt to the extremely heterogeneous browser landscape. On devices in the `high` cluster, smaller models reach above 3k tok/s during prefill and 100 tok/s during decode, while dropping to 65 tok/s during prefill and 30 tok/s during decode on the large `gemma4` model. Throughput on the `low` cluster is significantly lower, with decode in the range of 4–17 tok/s, but still shows that capable small models can run even in extremely constrained environments. Our results also show that while increased KV-cache depth has some effect on performance, LlamaWeb is able to scale to handle higher contexts while still achieving reasonable performance. For a more complete breakdown of model performance on every device in our study, see App. D.



(a) Prefill throughput.



(b) Decode throughput.

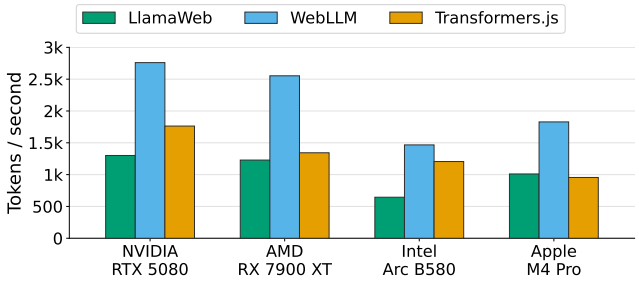
Figure 5: Throughput of the `llama` model across different llama.cpp backends and weight formats. The native backend is CUDA on the NVIDIA GPU, HIP on the AMD GPU, SYCL on the Intel GPU, and Metal on the Apple GPU.

6.2 Native Performance

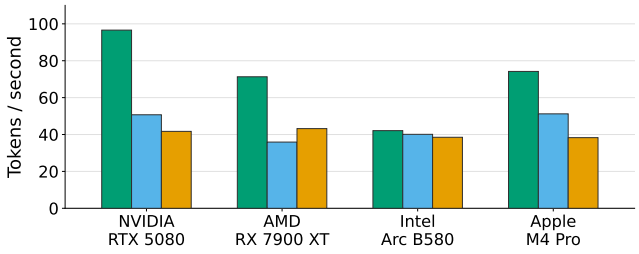
Along with the cross-device study, we also run llama.cpp natively, i.e., outside the browser, on four GPUs under several configurations as shown in Fig. 5 using llama.cpp's `llama-bench` tool. Native backends vary across devices, while Vulkan offers another cross-platform implementation for comparison against WebGPU. In the browser, WebGPU compilers add safety checks to avoid out-of-bounds buffer accesses and division-by-zero, so we also run the WebGPU backend with and without checks enabled gives a more comprehensive view of WebGPU performance. The WebGPU backend can also use the subgroup matrix feature when running natively through the Dawn WebGPU implementation, increasing the performance of our matrix multiplication and FlashAttention kernels.

As can be seen in Fig. 5a, the CUDA and Vulkan backends still significantly outperform WebGPU by up to 10× during prefill and 2.5× during decode across both model weight formats, likely due to better utilization of tensor cores and access to vendor-specific advanced features like asynchronous memory loads. Similarly, the Metal backend always outperforms the WebGPU backend, by over 2× during prefill and 50% during decode, although the difference is not as dramatic due to the lack of tensor cores and the lower overall performance on the smaller Apple GPU. We also run Vulkan on the Apple GPU using MoltenVK [32] and find that the WebGPU backend is up to 59% faster during prefill and 45% faster during decode. On the AMD GPU, the results vary; while the native HIP backend does always outperform the WebGPU backend, the Vulkan backend underperforms the WebGPU backend during prefill by 3×, but outperforms even the native HIP backend during decode by 38% on the `q4_k_m` model. On the Intel GPU, the WebGPU backend, with safety checks disabled, provides the most performant prefill on the `f16` model, beating the SYCL backend by 23%, and still outperforms the native SYCL backend by 10% during decode on the same model.

Overall, these results highlight that LlamaWeb is a competitive cross-platform implementation, matching and exceeding even native backend performance in some configurations, with room to improve in others. Importantly, it shows that cross-device tuning is an important part of building a portable inference engine, as LlamaWeb's tuned matrix multiplication parameters allow it to outperform the Vulkan-specific backend on some devices, despite the fact that Vulkan is the underlying runtime for WebGPU on Linux systems. Comparing performance with and without safety checks reveals that they cause an average 14% and 23% slowdown during prefill on the `q4_k_m` and `f16` models respectively, and a 5% and 1% slowdown during decode. These slowdowns, which reached up to 42% during prefill on the NVIDIA RTX 5080, show that there are opportunities for safe WebGPU to improve its performance, e.g., through better static analysis of memory accesses to remove runtime bounds-checks.

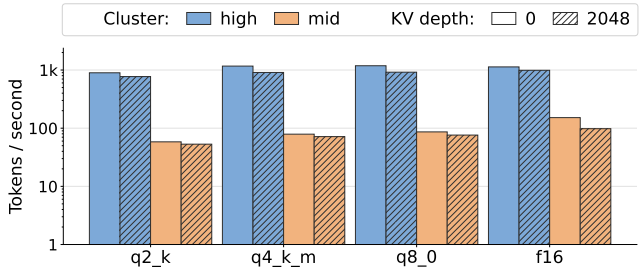


(a) Prefill throughput.

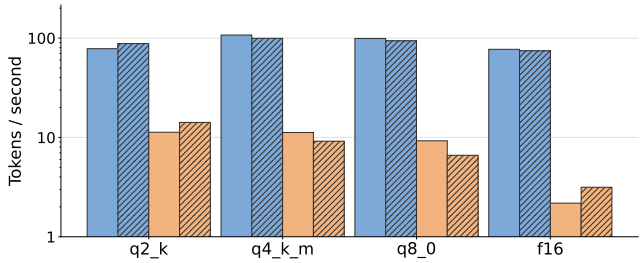


(b) Decode throughput.

Figure 6: Throughput comparison for the `llama` model with `f16` weights across web frameworks on Chrome. The operating system is macOS for the Apple GPU, Windows for the NVIDIA GPU, and Linux for the Intel and AMD GPUs.



(a) Prefill throughput (512-token prompt).



(b) Decode throughput (128 generated tokens).

Figure 7: Throughput on the `llama` model across four weight formats (`q2_k`, `q4_k_m`, `q8_0`, `f16`), grouped by the same device clusters as in Sec. 6.1.

6.3 Framework Comparison

We evaluated LlamaWeb performance against other frameworks by recording prefill and decode throughput on the `llama` model with `f16` weights on Chrome on the same 4 GPUs as Sec. 6.2, with results shown in Fig. 6. Because we did not have fine-grained control over benchmarking other frameworks, we used a consistent large prompt to measure prefill and a prompt that led to a relatively long decode stage to collect all measurements[7].

We calculate the geometric mean of normalized throughput across devices, as this avoids biasing devices with higher absolute throughput and correctly weights ratios. Overall, LlamaWeb achieves approximately 54% higher decode throughput than WebLLM and 69% higher decode throughput than Transformers.js, highlighting how our optimizations combine to provide state-of-the-art browser inference performance. However, prefill performance lags behind, with LlamaWeb achieving only 49% of WebLLM's throughput and 79% of Transformers.js's throughput.

The prefill gap can most likely be explained by: (1) WebLLM's use of TVM's sophisticated kernel fusion techniques to bring down the costs of loading model weights repeatedly for different small operations, and (2) Transformers.js's matrix multiplication subgroup-optimized implementation. In contrast to Transformers.js, LlamaWeb currently relies on a more portable, but less efficient register tiling matrix multiplication implementation in the browser, while our more performant kernel relying on the newer subgroup matrix feature can only run natively. In fact, using the prefill numbers from running LlamaWeb natively with checks from Sec. 6.2, LlamaWeb outperforms the prefill numbers from other frameworks on every device except WebLLM on the Apple M4 Pro, with a geometric mean speedup of 88% over WebLLM and 205% over Transformers.js. Therefore, although subgroup matrices are not yet available in browsers, LlamaWeb is in a strong position to provide competitive prefill performance once the feature reaches general availability. Implementing kernel fusion in LlamaWeb is also an exciting area for future work.

[7]For the exact prompts used, see App. E.

7 Quantization Performance

To characterize how our templated dequantization kernels scale across bit widths, we run the `llama` model on the `high` and `mid` clusters from Sec. 6.1 across four model weight formats: `q2_k`, `q4_k_m`, `q8_0`, and `f16` (Fig. 7). The `low` cluster is excluded from this study because the larger formats exceed Safari's iOS tab-memory budget.

As a memory-bound operation, decode throughput (Fig. 7b) shows a clear speedup when moving from `f16` to `q8_0` weights, e.g., 20% on the `high` cluster and 53% on the `mid` cluster with a KV-cache depth of 0. However, performance does not scale linearly when moving to lower-bit quantization formats; while the lower memory traffic does lead to an 8% increase in throughput when moving from `q8_0` to `q4_k_m` weights on `mid` cluster, moving from `q4_k_m` to `q2_k` weights causes a 17% decrease in throughput on the same cluster. Prefill (Fig. 7a) shows a flatter profile, and the `f16` weight format actually performs best on the `mid` cluster, with a median throughput of 152 tok/s vs. 79 for `q4_k_m`. Generally, performance across model weight format stays within an order

of magnitude, and the gap between the fastest and slowest model weight format for prefill is only 25% on the `high` cluster.

KV-cache growth has a similar effect across formats. Prefill drops 10–20% from depth 0 to 2048 on the `high` cluster regardless of weight format, consistent with the attention computation becoming proportionally larger. Decode is barely affected: median `q4_k` throughput only falls from 107 to 99 tok/s on the `high` cluster, and the other formats track within a few percent. Once a model weight format is chosen, KV-cache scaling behavior is fairly predictable.

These results show that as it stands llama.cpp quantization in WebGPU is mainly a technique for reducing the memory requirements for running a given model, rather than a performance optimization. Dequantization routines for generic model weight formats are complex and computationally heavy compared to hardware-native formats like `nvfp4`, which WebGPU does not support, and the current layouts may not be optimized for performance on the variety of GPUs that WebGPU targets. However, the results do validate that the templated kernels introduced in this work deliver competitive performance across various formats, and lay a foundation for future optimization work. For a more complete breakdown of model weight format performance across devices, see App. F.

8 Related Work

LLM Inference Engines. In the browser, WebLLM and Transformers.js provide WebGPU-based inference through MLC-LLM and ONNX Runtime, and we compare against these frameworks in our evaluation. WeInfer [6] improves aspects of WebLLM's execution scheduling, but does not target the same broader goals around memory efficiency, performance portability, and quantization support explored in this work. MediaPipe [39] also supports WebGPU inference, though current examples and documentation focus primarily on Gemma-family models. Outside the browser, systems such as MLX [23], vLLM [34], and TensorRT-LLM [51] target native GPU backends and high-throughput server inference, often emphasizing batching and datacenter deployment rather than execution on consumer and edge devices.

Performance Portability. The existing browser-based inference frameworks we evaluate do specialize some kernels based on device characteristics, but their limited performance portability strategies are generally undocumented. Beyond WebGPU, prior work has formalized the notion of performance portability and proposes quantitative metrics for measuring performance gaps [54]. Portable GPU libraries such as CLBlast expose numerous tuning parameters per-kernel and maintain community-contributed databases of kernel performance [49]. Auto-tuning frameworks like GPTune explore large optimization spaces for parallel applications and leverage performance data collected from many machines to guide future tuning decisions [38]. These techniques are complementary to the design goals of the LlamaWeb kernel library, as it can evolve to support more sophisticated tuning strategies and integration with kernel tuning frameworks.

Quantization. Quantization approaches can be broken down into training and post-training methods. In quantization-aware training, weights are learned while taking into account quantization effects [7, 12, 29]. Post-training quantization (PTQ) includes weight-only methods such as GPTQ [13], which does substantial offline analysis and optimization while modifying weights, and AWQ [36], which does a more lightweight statistics-based analysis. Runtime-aware PTQ methods like SmoothQuant [66] and LLM.int8() [10] combine offline analysis with runtime computation like activation scaling or mixed-precision execution. Quantization work is often based around developing new quantization strategies, rather than integrating them into a single runtime and templated kernels like this work. Support for various model weight formats and developing extensible kernel formats like we do in this work is an important step towards the development and deployment of novel quantization techniques.

9 Future Work

This work is only the starting point for LlamaWeb and inference in the browser using WebGPU. Within LlamaWeb itself, interesting future work opportunities include: (1) advances in performance-portable tuning techniques to enable maximal WebGPU performance on diverse systems, (2) new techniques for handling the functional and performance requirements of diverse quantization formats, (3) dynamic kernel fusion to reduce the overhead of repeatedly loading memory and launching many small GPU kernels, and (4) support for mixture-of-expert and multi-modal models in WebGPU (with several llama.cpp contributors already making strides in this direction).

There are also opportunities to optimize WebGPU implementations and improve the specification itself to better support applications like LLM inference in the browser, including: (1) static analysis techniques and optimizations to avoid safety and bounds-checks, which decrease performance in the browser, (2) more robust language-level handling of floating-point differences across systems to ensure model stability under lower-precision execution, and (3) extensions to WebGPU to support new data-types, e.g., `u16`, that enable more efficient memory loading and native hardware acceleration.

10 Conclusion

In this work, we introduce LlamaWeb, a WebGPU backend for llama.cpp that is designed with goals of memory efficiency, performance portability, and broad model weight format support in mind. We show that compared to existing frameworks, LlamaWeb provides state-of-the-art results with respect to these goals. LlamaWeb provides a new opportunity for accessible and performant browser-based LLM inference, with several outside contributors already contributing to and utilizing it for exciting new use-cases. We look forward to seeing how the combined llama.cpp and WebGPU ecosystems continue to evolve based upon the foundation laid in this work.

11 Acknowledgments

We thank the maintainers and developers of llama.cpp for their support in helping us integrate the WebGPU backend into llama.cpp. We also thank the outside contributors who are already contributing to and improving the llama.cpp WebGPU backend. This work was supported in part by an NDSEG fellowship.

References

[1] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. 2025. SmolLM2: When Smol Goes Big – Data-Centric Training of a Small Language Model. arXiv:2502.02737 https://arxiv.org/abs/2502.02737

[2] Alexander Amini, Anna Banaszak, Harold Benoit, Arthur Böök, Tarek Dakhran, Song Duong, Alfred Eng, Fernando Fernandes, Marc Härkönen, Anne Harrington, Ramin Hasani, Saniya Karwa, Yuri Khrustalev, Maxime Labonne, Mathias Lechner, Valentine Lechner, Simon Lee, Zetian Li, Noel Loo, Jacob Marks, Edoardo Mosca, Samuel J. Paech, Paul Pak, Rom N. Parnichkun, Alex Quach, Ryan Rogers, Daniela Rus, Nayan Saxena, Bettina Schlager, Tim Seyde, Jimmy T. H. Smith, Aditya Tadimeti, and Neehal Tumma. 2025. LFM2 Technical Report. arXiv:2511.23404 https://arxiv.org/abs/2511.23404

[3] Apple Inc. 2026. Metal. https://developer.apple.com/documentation/metal/.

[4] Elie Bakouch, Carlos Miguel Patiño, Anton Lozhkov, Edward Beeching, Aymeric Roucher, Nouamane Tazi, Aksel Joonas Reedi, Guilherme Penedo, Hynek Kydlicek, Clémentine Fourrier, Nathan Habib, Kashif Rasul, Quentin Gallouédec, Hugo Larcher, Mathieu Morlon, Joshua Lochner, Vaibhav Srivastav, Xuan-Son Nguyen, Colin Raffel, Lewis Tunstall, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. 2025. SmolLM3: Smol, Multilingual, Long-Context Reasoner. https://huggingface.co/blog/smollm3.

[5] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small Language Models are the Future of Agentic AI. arXiv:2506.02153 https://arxiv.org/abs/2506.02153

[6] Zhiyang Chen, Yun Ma, Haiyang Shen, and Mugeng Liu. 2025. WeInfer: Unleashing the Power of WebGPU on LLM Inference in Web Browsers. In *Proceedings of the ACM on Web Conference 2025*. Association for Computing Machinery. https://doi.org/10.1145/3696410.3714553

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Training Deep Neural Networks with Low Precision Multiplications. arXiv:1412.7024 https://arxiv.org/abs/1412.7024

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*. https://doi.org/10.48550/arXiv.2205.14135

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2023. Flash-Decoding for Long-Context Inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

[10] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. arXiv:2208.07339 https://arxiv.org/abs/2208.07339

[11] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. arXiv:2305.14314 https://arxiv.org/abs/2305.14314

[12] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. 2020. Learned Step Size Quantization. arXiv:1902.08153 https://arxiv.org/abs/1902.08153

[13] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 https://arxiv.org/abs/2210.17323

[14] Georgi Gerganov et al. 2026. llama.cpp: Inference of LLaMA models in pure C/C++. https://github.com/ggml-org/llama.cpp.

[15] Allen Gersho and Robert M. Gray. 1991. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers.

[16] Google. 2026. Dawn: A WebGPU Implementation. https://dawn.googlesource.com/dawn.

[17] Google. 2026. Protocol Buffers Documentation. https://protobuf.dev/.

[18] Google Cloud. 2019. BFloat16: The Secret to High Performance on Cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[19] Google DeepMind. 2026. Gemma 4 Model Card. https://ai.google.dev/gemma/docs/core/model_card_4.

[20] Khronos Group. 2026. Vulkan 1.3 Specification. https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html.

[21] Albert Gu and Tri Dao. 2024. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. arXiv:2312.00752 https://arxiv.org/abs/2312.00752

[22] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. *SIGPLAN Not.* (2017). doi:10.1145/3140587.3062363

[23] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. 2026. *MLX: Efficient and Flexible Machine Learning on Apple Silicon*. https://github.com/ml-explore

[24] Hugging Face. 2026. *Transformers.js*. https://github.com/huggingface/transformers.js

[25] Hugging Face. 2026. Transformers.js Documentation. https://huggingface.co/docs/transformers.js/index.

[26] Hugging Face. 2026. *Transformers.js Examples*. https://github.com/huggingface/transformers.js-examples

[27] Erik Johannes Husom, Arda Goknil, Merve Astekin, Lwin Khin Shar, Andre KÃ¥sen, Sagar Sen, Benedikt Andreas Mithassel, and Ahmet Soylu. 2025. Sustainable LLM Inference for Edge AI: Evaluating Quantized LLMs for Energy Efficiency, Output Accuracy, and Inference Latency. *ACM Trans. Internet Things* (2025). https://doi.org/10.1145/3767742

[28] IBM. 2026. Granite Models Documentation. https://www.ibm.com/granite/docs/models/granite.

[29] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. arXiv:1712.05877 https://arxiv.org/abs/1712.05877

[30] Nidhal Jegham, Marwan Abdelatti, Chan Young Koh, Lassad Elmoubarki, and Abdeltawab Hendawi. 2025. How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference. arXiv:2505.09598 https://arxiv.org/abs/2505.09598

[31] Iwan Kawrakow. 2023. K-Quants. https://github.com/ggml-org/llama.cpp/pull/1684#issuecomment-2474462323. GitHub comment.

[32] Khronos Group. 2026. *MoltenVK*. https://github.com/KhronosGroup/MoltenVK Vulkan portability implementation over Apple's Metal API. Accessed: 2026-05-11.

[33] Jennifer King, Kevin Klyman, Emily Capstick, Tiffany Saade, and Victoria Hsieh. 2025. User Privacy and Large Language Models: An Analysis of Frontier Developers' Privacy Policies. arXiv:2509.05382 https://arxiv.org/abs/2509.05382

[34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. https://doi.org/10.1145/3600006.3613165

[35] Reese Levine. 2026. PreWGSL: Universal preprocessor for WGSL shaders. https://github.com/reeselevine/pre-wgsl.

[36] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2026. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv:2306.00978 https://arxiv.org/abs/2306.00978

[37] Alexander H. Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, Alexandre Sablayrolles, Amélie Héliou, Amos You, Andy Ehrenberg, Andy Lo, Anton Eliseev, Antonia Calvi, Avinash Sooriyarachchi, Baptiste Bout, Baptiste Rozière, Baudouin De Monicault, Clémence Lanfranchi, Corentin Barreau, Cyprien Courtot, Daniele Grattarola, Darius Dabert, Diego de las Casas, Elliot Chane-Sane, Faruk Ahmed, Gabrielle Berrada, Gaëtan Ecrepont, Gauthier Guinet, Georgii Novikov, Guillaume Kunsch, Guillaume Lample, Guillaume Martin, Gunshi Gupta, Jan Ludziejewski, Jason Rute, Joachim Studnia, Jonas Amar, Joséphine Delas, Josselin Somerville Roberts, Karmesh Yadav, Khyathi Chandu, Kush Jain, Laurence Aitchison, Laurent Fainsin, Léonard Blier, Lingxiao Zhao, Louis Martin, Lucile Saulnier, Luyu Gao, Maarten Buyl, Margaret Jennings, Marie Pellat, Mark Prins, Mathieu Poirée, Mathilde Guillaumin, Matthieu Dinot, Matthieu Futeral, Maxime Darrin, Maximilian Augustin, Mia Chiquier, Michel Schimpf, Nathan Grinsztajn, Neha Gupta, Nikhil Raghuraman, Olivier Bousquet, Olivier Duchenne, Patricia Wang, Patrick von Platen, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Pavankumar Reddy Muddireddy, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Quentin Torroba, Romain Sauvestre, Roman Soletskyi, Rupert Menneer, Sagar Vaze, Samuel Barry, Sanchit Gandhi, Siddhant Waghjale, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Teven Le Scao, Théo Cachet, Theo Simon Sorg, Thibaut Lavril, Thiziri Nait Saada, Thomas Chabal, Thomas Foubert, Thomas Robert, Thomas Wang, Tim Lawson, Tom Bewley, Tom Bewley, Tom Edwards, Umar Jamil, Umberto Tomasini, Valeriia Nemychnikova, Van Phung, Vincent Maladière, Virgile Richard, Wassim Bouaziz, Wen-Ding Li, William Marshall, Xinghui Li, Xinyu Yang, Yassine El Ouahidi, Yihan Wang, Yunhao Tang, and Zaccharie Ramzi. 2026. Ministral 3. arXiv:2601.08584 https://arxiv.org/abs/2601.08584

[38] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery. doi:10.1145/3437801.3441621

[39] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. 2019. MediaPipe: A Framework for Building Perception Pipelines. arXiv:1906.08172 https://arxiv.org/abs/1906.08172

[40] Rust Graphics Mages. 2026. wgpu. https://github.com/gfx-rs/wgpu.

[41] Meta. 2024. Llama 3.2 Model Card. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_2/.

[42] Microsoft. 2026. DirectX Specifications. https://microsoft.github.io/DirectX-Specs/.

[43] Microsoft. 2026. *ONNX Runtime.* https://github.com/microsoft/onnxruntime

[44] Microsoft. 2026. ONNX Runtime Web: Tutorials and Documentation. https://onnxruntime.ai/docs/tutorials/web/.

[45] MLC AI. 2026. WebLLM Chat Demo. https://chat.webllm.ai/.

[46] MLC team. 2026. *MLC-LLM.* https://github.com/mlc-ai/mlc-llm

[47] Mozilla. 2026. *Origin Private File System.* https://developer.mozilla.org/en-US/docs/Web/API/File_System_API/Origin_private_file_system

[48] Xuan-Son Nguyen. 2026. wllama: Run llama.cpp models in the browser. https://github.com/ngxson/wllama.

[49] Cedric Nugteren. 2018. CLBlast: A Tuned OpenCL BLAS Library. In *Proceedings of the International Workshop on OpenCL (IWOCL '18).* ACM. doi:10.1145/3204919.3204924

[50] NVIDIA. 2025. Introducing NVFP4 for Efficient and Accurate Low-Precision Inference. https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/.

[51] NVIDIA. 2026. *TensorRT-LLM.* https://github.com/NVIDIA/TensorRT-LLM

[52] Open Compute Project. 2023. OCP Microscaling Formats (MX) Specification Version 1.0. https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf.

[53] OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 https://arxiv.org/abs/2508.10925

[54] S. J. Pennycook, J. D. Sewall, and V. W. Lee. 2016. A Metric for Performance Portability. arXiv:1611.07409 https://arxiv.org/abs/1611.07409

[55] PrismML. 2025. 1-bit Bonsai 8B Whitepaper. https://github.com/PrismML-Eng/Bonsai-demo/blob/main/1-bit-bonsai-8b-whitepaper.pdf. Technical report.

[56] Qwen Team. 2026. Qwen3.5-2B. https://huggingface.co/Qwen/Qwen3.5-2B.

[57] Charlie F. Ruan, Yucheng Qin, Akaash R. Parthasarathy, Xun Zhou, Ruihang Lai, Hongyi Jin, Yixin Dong, Bohan Hou, Meng-Shiun Yu, Yiyan Zhai, Sudeep Agarwal, Hangrui Cao, Siyuan Feng, and Tianqi Chen. 2026. WebLLM: A High-Performance In-Browser LLM Inference Engine. arXiv:2412.15803 https://arxiv.org/abs/2412.15803

[58] Jon Saad-Falcon, Avanika Narayan, Hakki Orhun Akengin, J. Wes Griffin, Herumb Shandilya, Adrian Gamarra Lafuente, Medhya Goel, Rebecca Joseph, Shlok Natarajan, Etash Kumar Guha, Shang Zhu, Ben Athiwaratkun, John Hennessy, Azalia Mirhoseini, and Christopher Ré. 2026. Intelligence per Watt: Measuring Intelligence Efficiency of Local AI. arXiv:2511.07885 https://arxiv.org/abs/2511.07885

[59] Sha Sajadieh, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, Lapo Santarlasci, Juan Pava, Nestor Maslej, Russ Altman, Erik Brynjolfsson, Carla Brodley, Jack Clark, Virginia Dignum, Vipin Kumar, James Landay, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Elham Tabassi, Russell Wald, Toby Walsh, and Dan Weld. 2026. The AI Index 2026 Annual Report. https://hai.stanford.edu/assets/files/ai_index_report_2026.pdf

[60] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 Technical Report. arXiv:2503.19786 https://arxiv.org/abs/2503.19786

[61] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. 2024. QuIP#: Even Better LLM Quantization with Hadamard Incoherence and Lattice Codebooks. arXiv:2402.04396 https://arxiv.org/abs/2402.04396

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17).* https://doi.org/10.48550/arXiv.1706.03762

[63] W3C. 2026. WebGPU. https://www.w3.org/TR/webgpu/.

[64] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* (2019). doi:10.1145/3360559

[65] WebKit Contributors. 2026. *WebKit: WebKit Browser Engine on GitHub.* https://github.com/WebKit/WebKit

[66] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2024. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. arXiv:2211.10438 https://arxiv.org/abs/2211.10438

[67] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 https://arxiv.org/abs/2505.09388

[68] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion.* Association for Computing Machinery. doi:10.1145/2048147.2048224

[69] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. arXiv:2307.13854 https://arxiv.org/abs/2307.13854

A Number of Available Models

We searched Hugging Face for available models for each framework at the time of submission. LlamaWeb loads models in the GGUF format, of which there were 177,691 on Hugging Face in May 2026. While not all of these models are likely to run in the browser due to large sizes or legacy formats, our tests and evaluation show that every recent model we tested did run successfully on large-enough GPUs. Transformers.js uses ONNX Runtime as its backend, and there were 41,632 ONNX models available on Hugging Face. As with LlamaWeb, not all of these models are likely to be compatible with Transformers.js, but to provide a fair comparison we use this number in Tab. 1. WebLLM only directly supports models in the MLC format, and the Hugging Face mlc-ai model repository currently contains 400 models in this format.

B Framework Model Weight Support

In this section we briefly discuss the model weight formats available for inference in each of the three frameworks we evaluate in this work. Within each framework, a single model and its activations may be stored in several different formats, depending on how the model is quantized. However, we do not count each of these potential mixtures separately, and focus only on the core underlying data formats.

LlamaWeb. Our implementation supports `f32` and `f16` floating-point values. As discussed in Sec. 2, WebGPU does not include support for some native floating-point formats. We also support the three main quantization families in llama.cpp: legacy, including `q4_0`, `q4_1`, `q5_0`, `q5_1`, and `q8_0`, K-quants, including `q2_k`, `q3_k`, `q4_k`, `q5_k`, and `q6_k`, and I-quants, including `iq1_s`, `iq1_m`, `iq2_xxs`, `iq2_xs`, `iq2_s`, `iq3_xxs`, `iq3_s`, `iq4_nl`, and `iq4_xs`. Finally, we recently added support for the new `q1_0` format, and an outside collaborator added support for `mxfp4`. There are several other formats supported by llama.cpp that the WebGPU does not yet support, but which may be able to be added in the future, including 64-bit weights and ternary quantization formats.

WebLLM. By investigating the MLC-LLM codebase [46], which powers WebLLM and uses TVM under the hood, and specifically its `quantization.py` file, we determined that WebLLM currently supports 6 formats, including `f32`, `f16`, `bf16`, `f8`, and 3 and 4-bit formats. To support hardware-native formats like `bf16` and `f8`, which are not available in WebGPU, MLC-LLM generates WGSL code that "legalizes" the weights at runtime through bit-interpretation into types like `f32` and `f16` that can be executed by WebGPU. In practice, e.g., in the hosted WebLLM examples [45], `f32`, `f16`, and 4-bit quantization seem to be the most widely deployed model weight formats.

Transformers.js. We investigated the Transformers.js [24] and ONNX Runtime [43] codebases to determine their support for different weight formats. There is no centralized list of quantization format for the ONNX Runtime WebGPU backend, but we determined that it likely supports `f32`, `f16`, signed and unsigned 8 and 4-bit types interpreted directly as floating point values, as well as block or tensor-wise 4 and 2-bit quantization. In Transformers.js, `dtypes.js` lists support for `f32`, `f16`, signed and unsigned 8-bit types, 8, 4, 2, and 1-bit quantization, and a special type called `bnb4`. To our knowledge, `bnb4` and 8-bit quantization do not have paths to the ONNX Runtime WebGPU backend, while 1-bit quantization is implemented by expanding weights at runtime into a 2-bit representation that is compatible with the ONNX Runtime WebGPU kernels. This means that while 1-bit models are still stored in a compact form, they incur a memory overhead at runtime. Overall, this leads us to conclude that 7 weight formats are supported for inference in WebGPU by Transformers.js.

In practice, we find that like WebLLM, the publicly available Transformers.js examples [26] tend to use `f32`, `f16`, and 4-bit quantization model weight formats.

Table 5: Cluster membership after k-means on log-throughput feature vectors with $k = 3$, as analyzed in Sec. 6.1.

Cluster	Devices
`high`	RTX 5080, RTX 5070, RTX 40-series, RX 7900 XT, Arc B580, M4 (32 GB), M3 (16 GB)
`mid`	Iris Xe (iGPU), M2, Snapdragon X Elite, Galaxy S24
`low`	iPhone 17 Pro Max, iPhone 15, Adreno 7xx, Mali (Valhall), PowerVR D-series

C Clustering GPUs with k-Means

To understand how LlamaWeb performs on GPUs across vendors and device characteristics, we group the 16 GPUs in our evaluation in Sec. 6.1 into three clusters via k-means. This appendix documents how the feature vectors are constructed, how k was chosen, and which devices end up in each cluster (Tab. 5).

Feature Vectors. Each device contributes one row to the input matrix. Columns are the per-(model, phase, KV-depth) measurements collected during the cross-device study: 10 models × 2 phases (prefill, decode) × 2 KV-cache depths (0 and 2048) yields 40 feature columns. Cell values are $\log 1p(\text{throughput})$ in tokens/sec; the log transform normalizes throughput values which span more than three orders of magnitude. When a model hasn't run on a device (e.g. `gemma4` on an iPhone with a tight tab-memory cap), we impute the missing cell with the per-column median across all devices that did run it.

Choice of k and Cluster Membership. We fit k-means with $k \in \{2, 3, 4, 5\}$ on the 16-device feature matrix. Inertia decreases without a sharp elbow ($358 \to 272 \to 223 \to 173$), highlighting the inherent heterogeneity of devices supported by WebGPU. For analysis in this paper, we set $k = 3$, as it leads to an intuitive split of devices as shown in Tab. 5. The `high` cluster captures the high-end discrete GPUs (RTX 5080/5070/40-series, RX 7900 XT, Arc B580) and the higher-tier Apple silicon (M3, M4). The `mid` cluster collects integrated and high-end-mobile GPUs (Iris Xe, M2, Snapdragon X Elite, Galaxy S24). The `low` cluster contains iOS devices and the low-power Android GPUs (Adreno 7xx, Mali (Valhall), PowerVR D-series, iPhone 15, iPhone 17 Pro Max).

D Per-Device Portability Results

This appendix supplements Sec. 6.1 with full per-device data for the portability study. Figure 8a shows the coverage and prefill throughput across each model and device in our study, while Fig. 8b shows the decode throughput. Dark blue cells correspond to higher throughput, light blue to lower throughput, and cells where the given model-device combo didn't run are left blank.

Each sub-figure in Fig. 9 and Fig. 10 reports prefill and decode throughput in bar graph form for a single model across every device on which it ran. Bars are colored by device family (NVIDIA, AMD, Intel, Apple-Mac, Apple-iOS, Qualcomm, Samsung, ARM, Imagination Technologies), with a hatch pattern distinguishing families that share a color slot. Within each device, paired bars encode KV-cache depth: solid fill for depth 0 and the same color faded for depth 2048. Models are presented in order of parameter count.

On a couple devices, e.g., the Arm Mali and Imagination Technologies PowerVR GPUs, the `lfm`, `gemma3`, `qwen3`, and `qwen3.5` models ran successfully at a KV-cache depth of 0, but crashed at a KV-cache depth of 2048. In these cases, we include the successfully collected data from the KV-cache of 0 in our clustering and analysis.

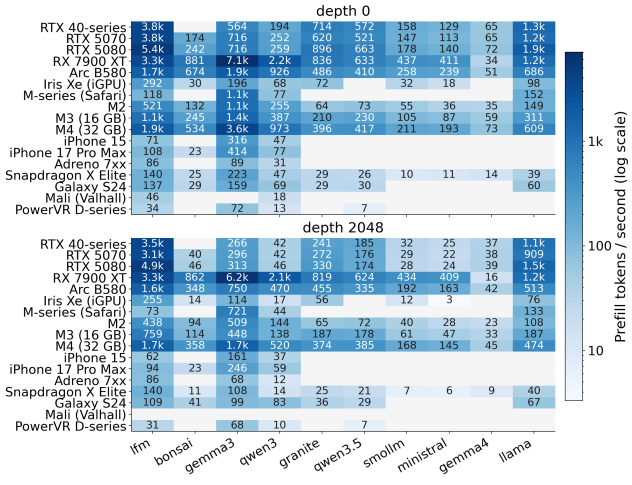
E Framework Comparison Prompts

These are the prompts we used to measure prefill and decode performance across different WebGPU inference frameworks for our evaluation in Sec. 6.3:

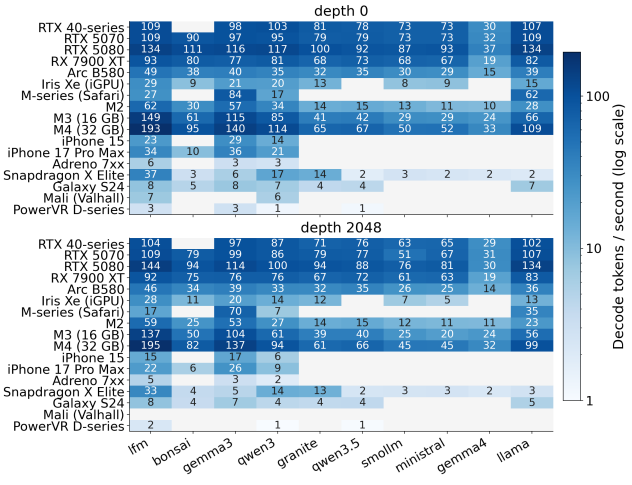
- **Decode:** *Write a story about a turtle.*
- **Prefill:** *Summarize this story about a turtle. In a small pond nestled among the tall reeds of a lush forest, a tiny turtle named Terry lived a simple life. He spent his days swimming in the pond, chasing after the occasional fish, and basking in the warm sun on a rock near the water's edge. Terry was a bit of an oddity among his fellow turtles, and he had a thirst for adventure. He longed to explore the world beyond his pond, to discover new lands, and to meet new creatures. One day, a strong storm rolled in, bringing heavy rain and powerful winds. The pond began to flood, and Terry found himself swept away by the rushing water. He tumbled through the air, his shell rattling against the rocks, and landed with a splash in a nearby stream. As he struggled to swim back to the pond, Terry spotted a small wooden boat drifting downstream. The boat was old and weathered, but it looked sturdy enough to carry him to safety. Without hesitation, Terry scurried into the boat and began to paddle. The journey was long and arduous, but Terry persevered. He navigated through treacherous rapids and dodged snapping fish. Finally, after what seemed like an eternity, he spotted the familiar outline of his home pond. As he emerged from the boat, Terry was greeted by a group of friendly otters, who had been watching him from a distance. They welcomed him with open arms, and Terry was amazed by their kindness and generosity. The otters took Terry under their wing, teaching him how to catch fish and navigate the waters. They showed him the secret spots where the best food could be found, and they introduced him to their friends, a wise old beaver named Benny. Benny, it turned out, was a master of the forest. He had spent years building a network of hidden tunnels and secret passageways, which he used to transport his family and friends to safety during times of drought or flood. Terry was amazed by the complexity of the beaver's underground world and begged Benny to take him on a journey through the forest. Benny, seeing the excitement in Terry's eyes, agreed to take him on a journey. As they explored the forest together, Terry discovered a hidden world of creatures he had never seen before. There were rabbits with bright pink noses, squirrels with fluffy tails, and even a family of field mice who lived in a cozy little burrow. The journey was long and winding, but Terry was thrilled to be a part of Benny's secret world. He realized that there was more to life than just swimming in the pond and catching fish. He had discovered a new passion, one that would take him on many more adventures in the years to come. From that day on, Terry became known as the greatest turtle explorer in the forest. He continued to visit his friends and family, but he also began to explore the world beyond his pond. He discovered hidden waterfalls, secret meadows, and even a hidden cave or two. And though he still lived in his pond, Terry knew that he had found a new home, one that was full of adventure, friendship, and the thrill of discovery.*

F Per-Device Quantization Results

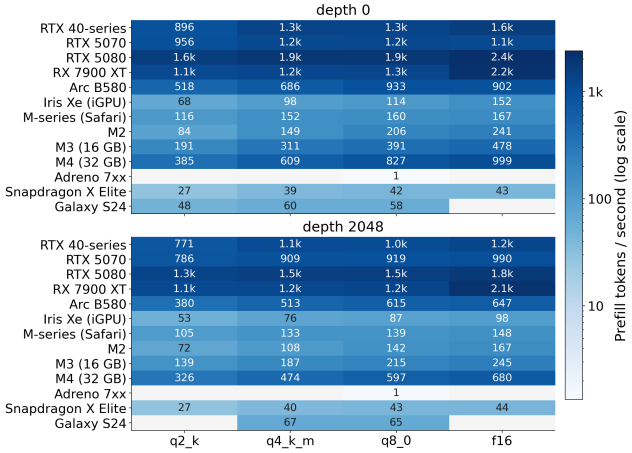
This appendix supplements Sec. 7 with full per-device data for the cross-quantization study. Similar to App. D, Fig. 8c and Fig. 8d show the coverage and prefill and decode throughput respectively of running the `llama` model with different weight formats across the devices in our study, and each sub-figure in Fig. 11 reports the throughput per-device and weight format in bar graph form.



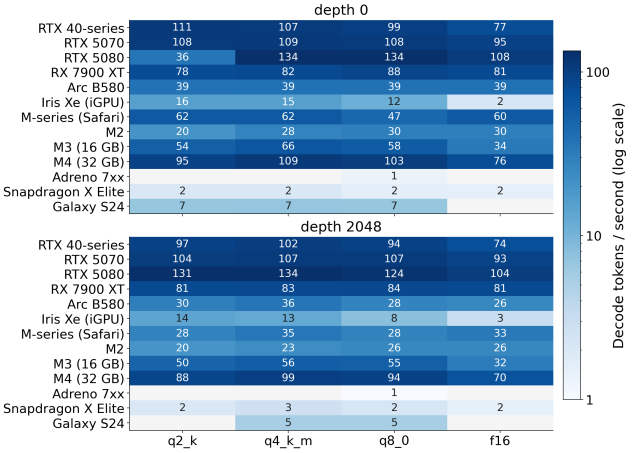
(a) Per-device prefill throughput across models.



(b) Per-device decode throughput across models.

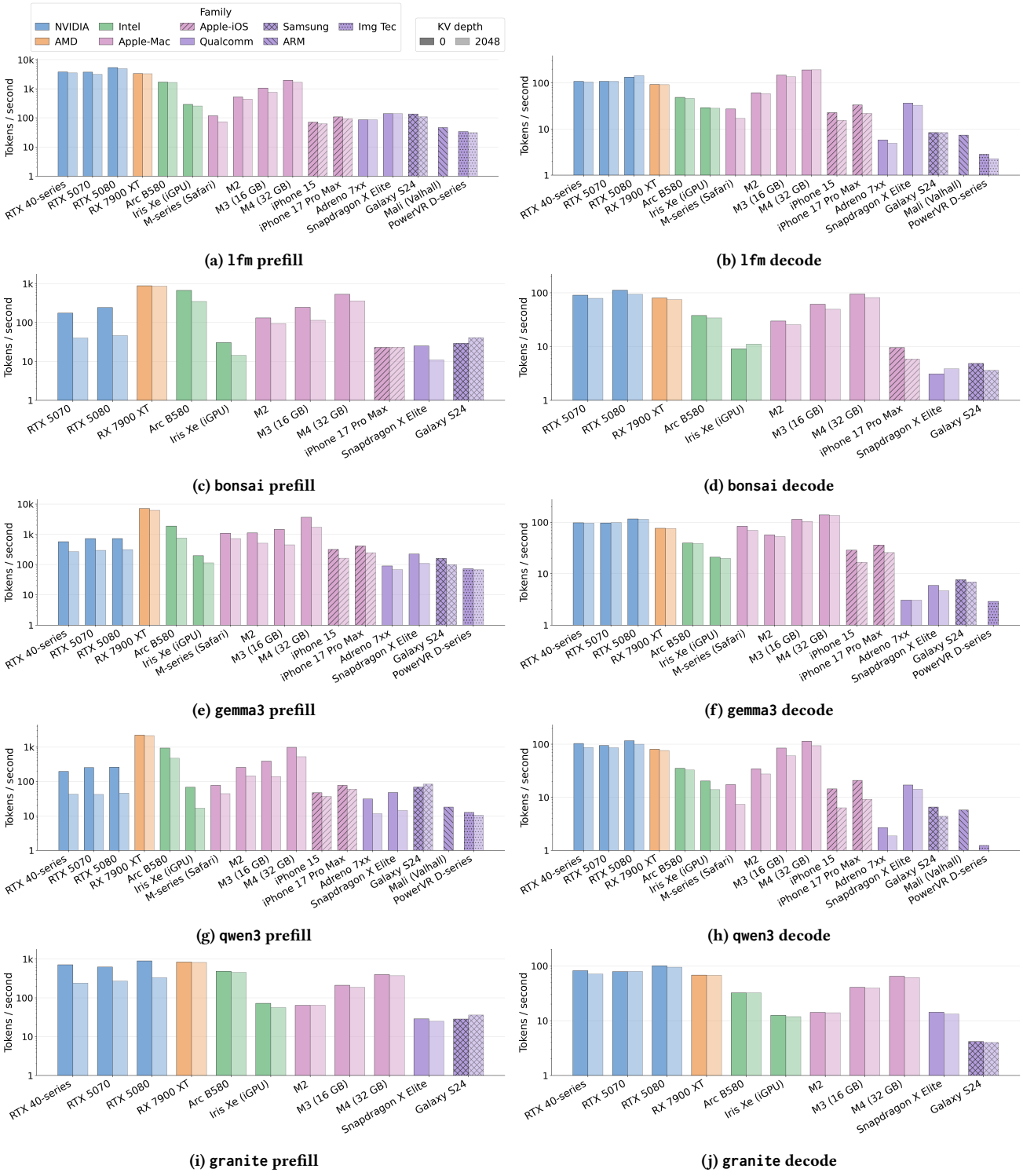


(c) Per-device prefill throughput across model weight formats.



(d) Per-device decode throughput across model weight formats.

Figure 8: Coverage matrices for the portability and cross-quantization studies. Each cell reports throughput on a shared log color scale; blank cells indicate that the model or quantization variant did not run on that device. Heatmaps within each panel show KV-cache depth 0 (top) and 2048 (bottom).



(a) `lfm` prefill

(b) `lfm` decode

(c) `bonsai` prefill

(d) `bonsai` decode

(e) `gemma3` prefill

(f) `gemma3` decode

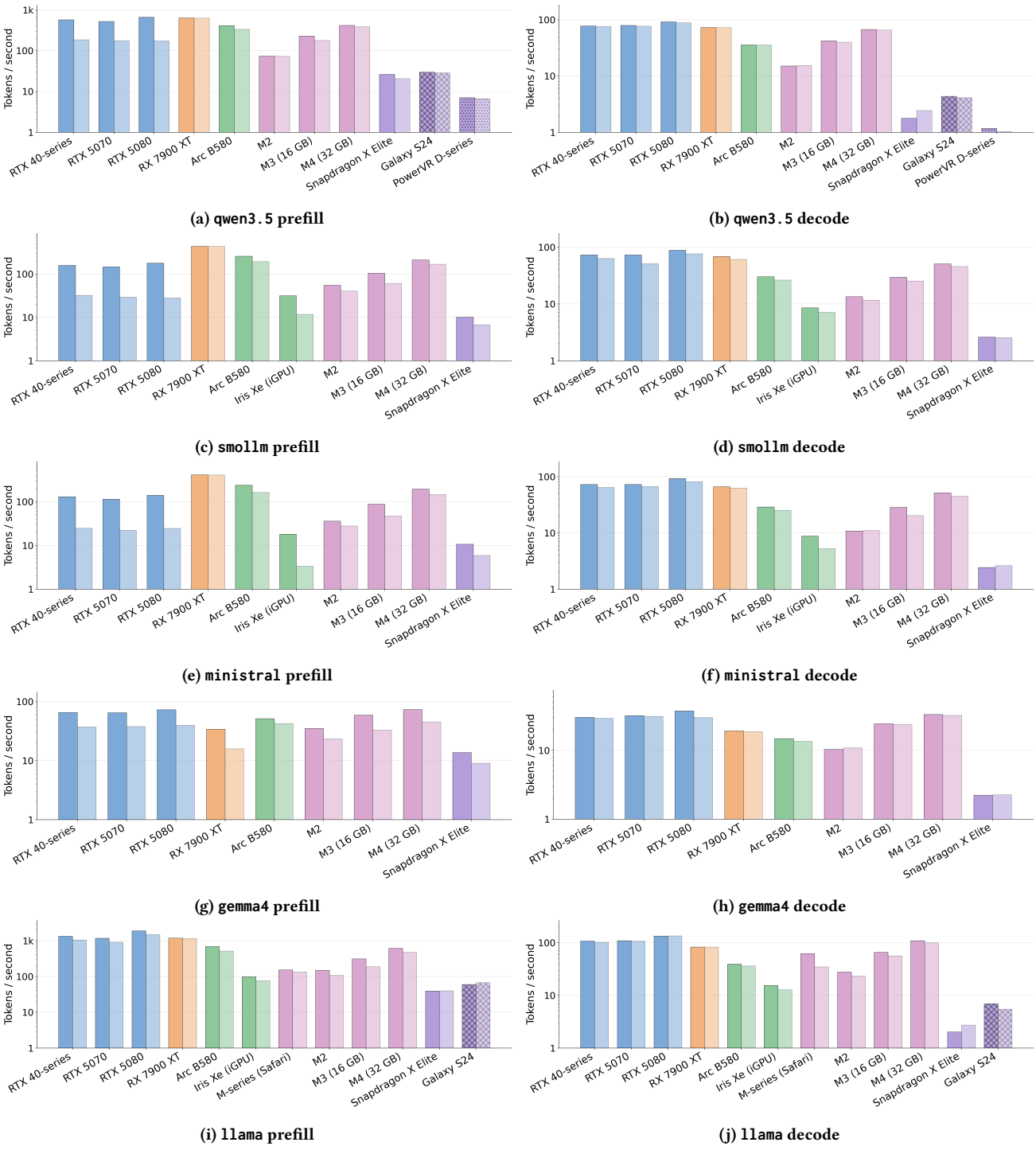
(g) `qwen3` prefill

(h) `qwen3` decode

(i) `granite` prefill

(j) `granite` decode

Figure 9: Per-device prefill and decode throughput by model, part 1 of 2.



(a) `qwen3.5` prefill

(b) `qwen3.5` decode

(c) `smollm` prefill

(d) `smollm` decode

(e) `ministral` prefill

(f) `ministral` decode

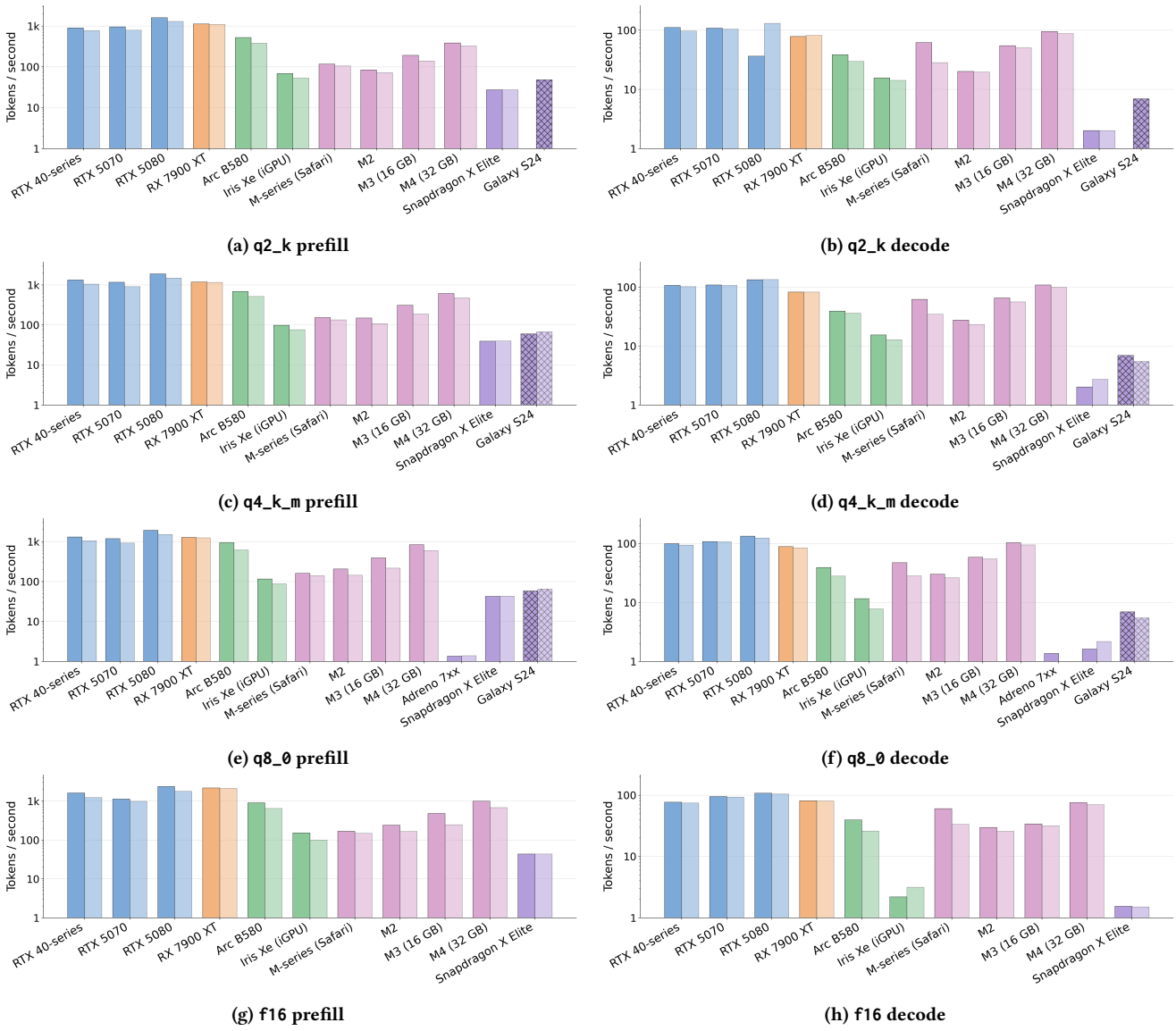
(g) `gemma4` prefill

(h) `gemma4` decode

(i) `llama` prefill

(j) `llama` decode

Figure 10: Per-device prefill and decode throughput by model, part 2 of 2.



(a) `q2_k` prefill

(b) `q2_k` decode

(c) `q4_k_m` prefill

(d) `q4_k_m` decode

(e) `q8_0` prefill

(f) `q8_0` decode

(g) `f16` prefill

(h) `f16` decode

Figure 11: Per-device prefill and decode throughput for `llama` under different model weight formats.